Weston Fiala
EEP524A - Final Project
12/13/18

Mandelbrot Generation

Overview

For my final project, I chose to implement a mandelbrot set generator. The mandelbrot set is a fractal that emerges from iterating over the function $Z_{n+1} = {z_n}^N + c$, where Z_{n+1} is the value of a point in the complex plane that is being calculated, Z_n is the current value at the point in the complex plane, N is the order of the mandelbrot, c is a constant value that is determined by the location in the complex plane that the calculation is being run at. A point is in the set if, its absolute value does not exceed a set threshold after a set number of iterations. Numbers in the set are represented by a black pixel, and numbers outside the set are represented by some other color.

Algorithm

The algorithm that I chose to implement is called the escape time algorithm. The escape time algorithm iterates over every point in the set and continually finds its next value until the value found exceeds a set threshold, or exceeds the maximum number of iterations. Once each pixel has exited the iterations, it will take the value that it exited the loop at, its count, and determine what color it should be. These colors can be anything, I chose to loop over red, green, and blue.

At this point, each pixel is a set color, but the image has bands of color where the exit counts line up. These bands of color do not look good, so I implemented a linear interpolation that smooths the color between bands. This interpolation uses the complex value at the time of exit and determines how close it was to being the next color and smooths between the two. This smoothed look is aesthetically pleasing.

OpenCL Parallelism

The mandelbrot set is an embarrassingly parallel problem and works exceptionally well within OpenCL. Every pixel in the viewport of the mandelbrot set can be calculated in parallel. In each pixel calculation there is a fair amount of linear code, finding the count, but a kernel can be spun up for every single pixel in the image.

Pseudocode

```
For each pixel (Px, Py) on the screen, do:
{
  x0 = scaled x coordinate of pixel (scaled to lie in the Mandelbrot X scale (-2.5, 1))
  y0 = scaled y coordinate of pixel (scaled to lie in the Mandelbrot Y scale (-1, 1))
  complex_constant = {x0,y0}
  complex_value = complex_constant
  iteration = 0
  max_iteration = 1000
  while ( abs(complex_value) < BAILOUT  AND  iteration < max_iteration ) {
    Complex_value = pow(complex_value, ORDER) + complex_constant
    iteration = iteration + 1
  }
  // Used to avoid floating point issues with points inside the set.
  if ( iteration < max_iteration ) {
    log_zn = log( abs(complex_value) )
    nu = log( log_zn / log(BAILOUT) ) / log(ORDER)
    iteration = iteration + 1 - nu
  }
  color1 = palette[floor(iteration)]
  color2 = palette[floor(iteration) + 1]
  // iteration % 1 = fractional part of iteration.
  color = linear_interpolate(color1, color2, iteration % 1)
  plot(Px, Py, color)
}
```

Theoretical Analysis

OPI calculation

The operational intensity of the algorithm for the order 2 mandelbrot is:

Reads: 2 (complex constant) + 1 (order) + 1 (escape) + 3 (num colors) = 7
Writes: 1 (pixel value) = 1
Memory = Reads + Writes: 4(bytes/float) * (8) = 32

Calculation: 5(r*r + i*i + sqrt)(complex absolute) + 1(count < max) + 4(r*r + r*i + i*r + i*i) (complex multiplies) + 2(r+r + i+i)(complex add) = 8 + 4 = 12

Total = Calculation / Memory:12/32 = 3/8

The maximum operational intensity of the algorithm is the minimum OPI multiplied by the maximum number of iterations.

Total = (2/11 + (ORDER-1)/11) * NUM_ITERATIONS

For the most common order, 2, and 1000 iterations the minimum and maximum OPI's are
Min = 3/8
Max = 3000/32 ~ 94

Speedup

When comparing the serial implementation to the OpenCL implementation, there are big opportunities for speed up. In the serial implementation, every pixel needs to processed one after the other, while in OpenCL you can process every single pixel at once.

The section of the code that limits speed up is the pixels that run until max iterations. In OpenCL, the theoretical max speed would be the time it takes to run one pixel through all of the iterations. I will call this value, PIXEL_MAX . Since all of the other kernels will have finished executing before PIXEL_MAX is reached, the whole process cannot be sped up anymore than the duration of PIXEL_MAX . In the serial implementation, you must run each pixel one at a time. Sometimes pixels will finish fast, and some times they will finish slow, but the average time they will exit will likely be in between 1 and PIXEL_MAX, likely PIXEL_MAX/2. The total time for serial would be ~NUM_PIXELS * PIXEL_MAX/2.

With these two variables in play you can estimate the possible speedup that you can achieve. (NUM_PIXELS * PIXEL_MAX / 2) / PIXEL_MAX ~= NUM_PIXELS/2 times faster.

There are two big factors for lowering this speed up. One is transfering the data to and from the GPU takes time, and the other is not being able to run all of the pixels at once due to hardware restrictions. For the GPU I am using, it can have 20 execution units, with a max group size of 256, this limits your maximum pixels per bunch to 5120. If you are producing an image at 1920x1080, you have 2073600 pixels. This means that you will have to run 405 batches of pixel processing. As for transferring data to and from the CPU, the OPI for this kernel is highly skewed towards compute limiting, so I will ignore it.

With the slowdown from the batches our theoretical speedup drops from NUM_PIXELS/2 to ~NUM_PIXELS/910, in the best case of 100% EU utilization. So, for a 1920x1080 image, you could get an improvement of ~2275x faster. If you can only achieve 50% EU utilization, that drops to around ~1137x faster.

Measured Analysis

In this section I will go over the results that I found, and the methods for how I found them. I will discuss both the OpenCL and the Serial implementations. More focus is put on the OpenCL results as that is what I was optimizing.

Profiling

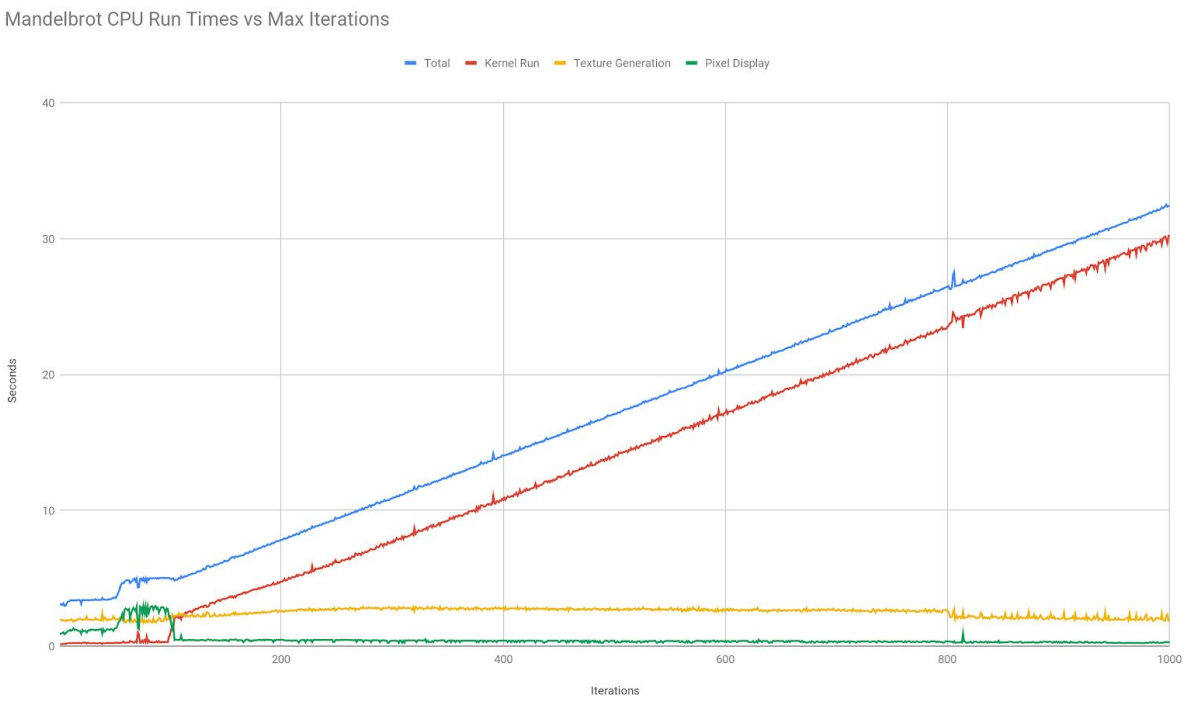
For my profiling I measured the time it took to do a variety of tasks. I looked at 9 specific measurements. Total runtime, bailout calculation, kernel enqueue, kernel run, texture generation, unmap buffer, pixel display, and overhead.

- Total: Time it takes for the entire program to run.
- Kernel Run: Time that the CPU is waiting for the kernel to finish.
- Texture Generation: Time to move the data from the kernel into the display buffer.
- Pixel Display: Time to render the pixels onto the screen.

With the combination of all these metrics, I was able to get a good look at how my kernel functioned.

Performance Results

I have included a variety of charts to help visualize the data that I collected. I have two sets of data, running the kernel with visuals enabled, and running the kernel with visuals disabled. The lower iteration counts performed much differently when you had visuals enabled/disabled. For each graph, the X axis is number of iterations, and the Y axis is time in seconds. Every data point in the graph is the total time it took to perform the given action over 100 frames.

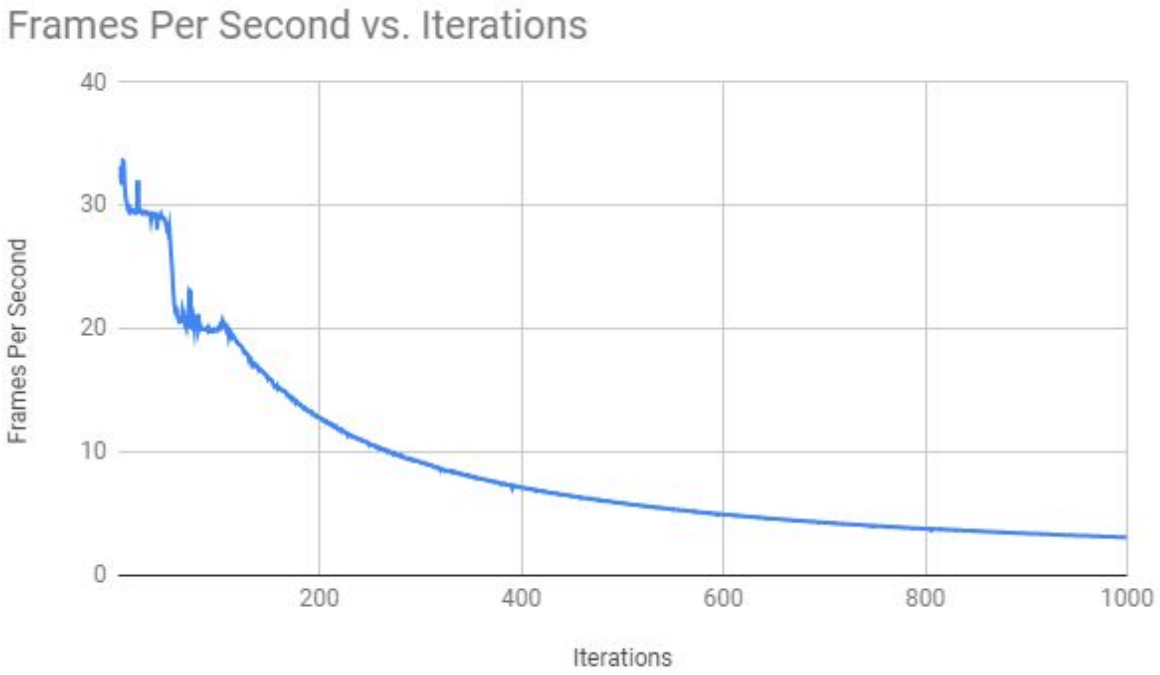


Performance results of running mandelbrot generator 1 - 1000 iterations, with display

Looking at the above graph you can draw two conclusions.

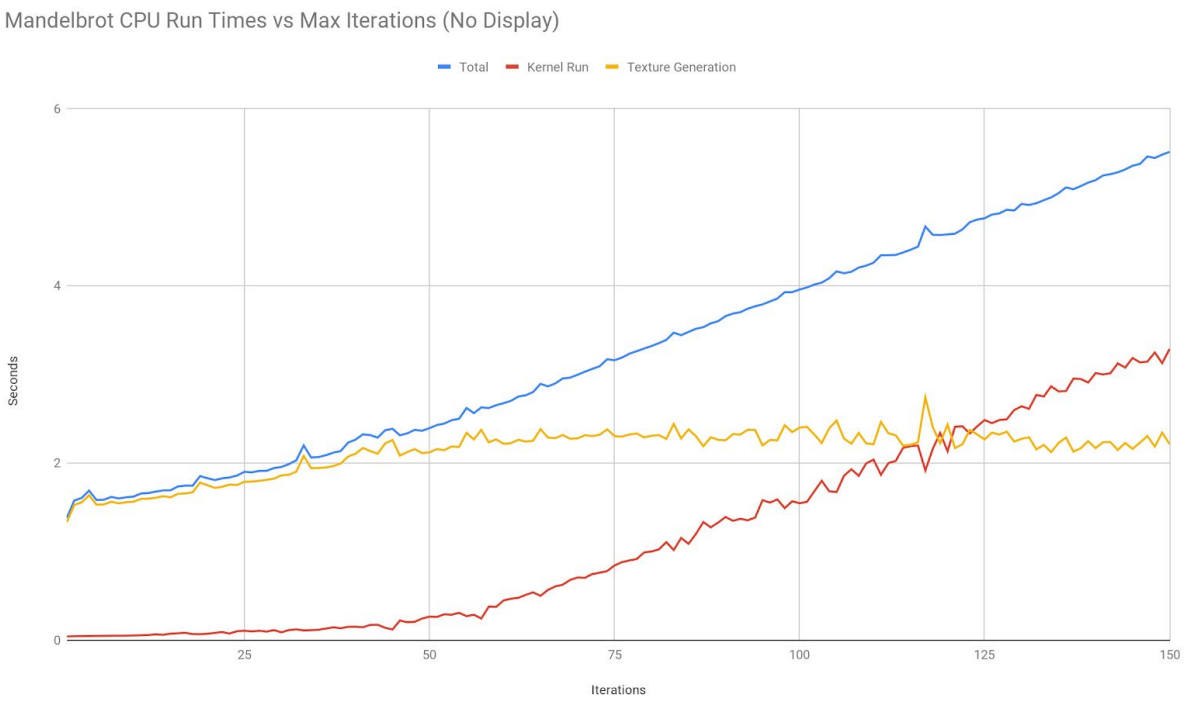
- The time to run the kernel increases linearly as you increase the iteration count.
- The time to copy the data from the kernel to the display is consistent.

With the easy to see conclusions down, there are two things of note to talk about with the beginning of the data. Displaying the pixels takes a long time at the beginning, but after a set point, becomes nearly trivial. The kernel seems to take zero execution time during the timeframe that the pixels take a long time to display. The reason for this odditiy is in the implementation of how I enqueue my kernels. Instead of running a kernel, retrieving the data, copying it over, displaying it, and repeat, I use a double buffer technique so that I can still do useful work on the GPU while I do processing on the CPU. As soon as one kernel is done running, and its data is available, I enqueue another kernel with a different data buffer, and then perform my CPU tasks. This allows the GPU to always have something to process. Using this technique, I was able to nearly double the framerate of lower iteration kernels. See below for FPS vs iterations.

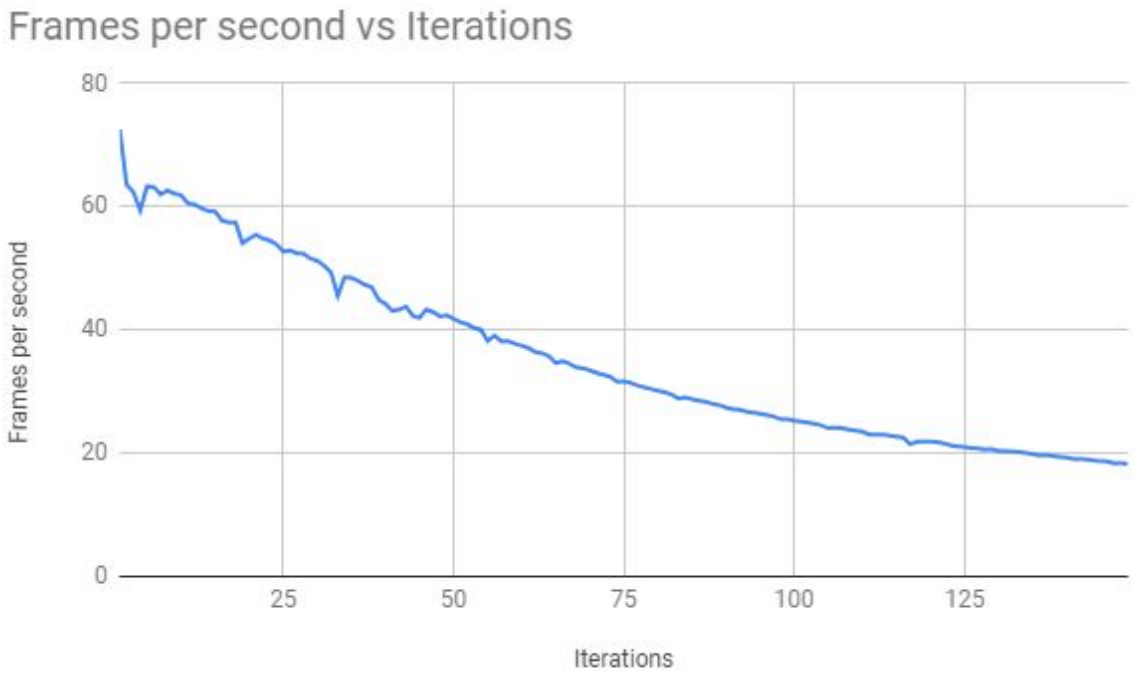


Frames per second vs iterations, with display

As you can see in the FPS graph, the oddities where pixel display take a long time distort the maximum frame rate that you can achieve. When you turn the display off, you can see what the kernel runtime looks like at the lower iteration counts (See Below). Past the point where you get odd display timings, the graph looks the same between display and no display. This is because at the higher iterations, the vast majority of time is spent waiting for the kernel to finish.



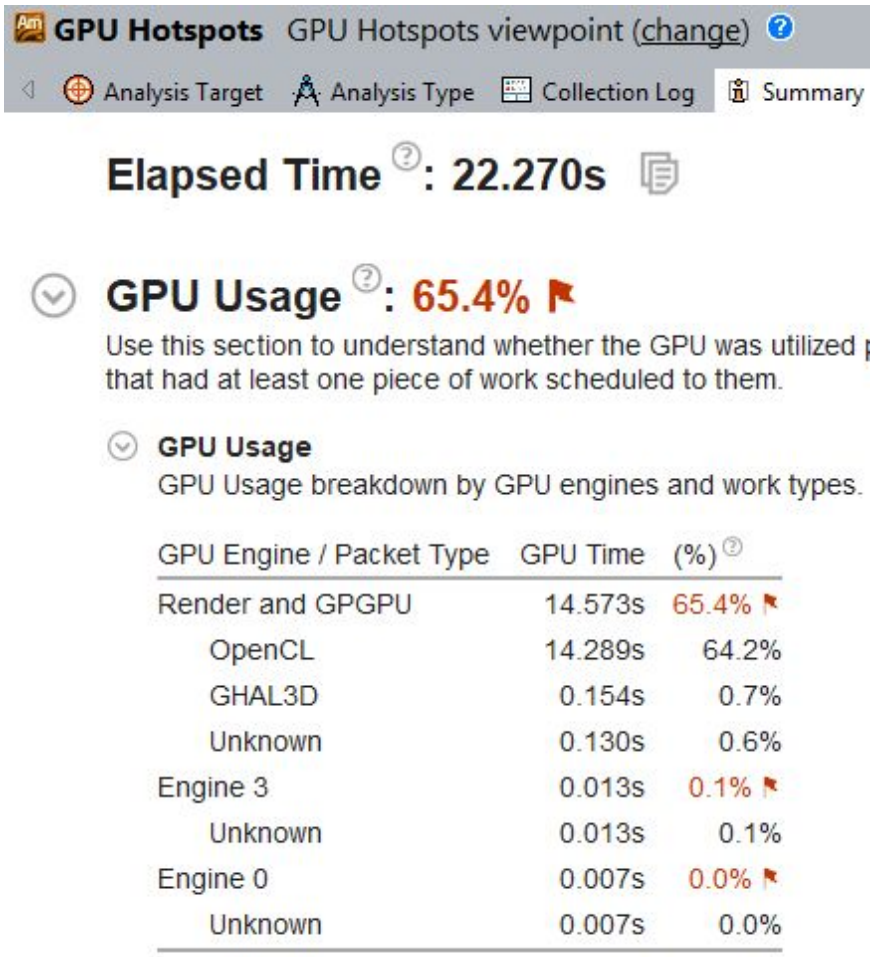
Performance results of running mandelbrot generator 1-150 iterations, no display



Frames per seconds vs iterations, no display

VTune

For my VTune analysis, I turned off the display and ran the program for 1000 frames at 100 iterations per frame. I found that, without finding a more efficient calculation to perform, I am at nearly maximum throughput on my kernel. The GPU is constantly at maximum capacity and is compute limited. Below are the statistics that VTune gave me after running various analysis modes. Below are the snips of my runs through VTune on my kernel.



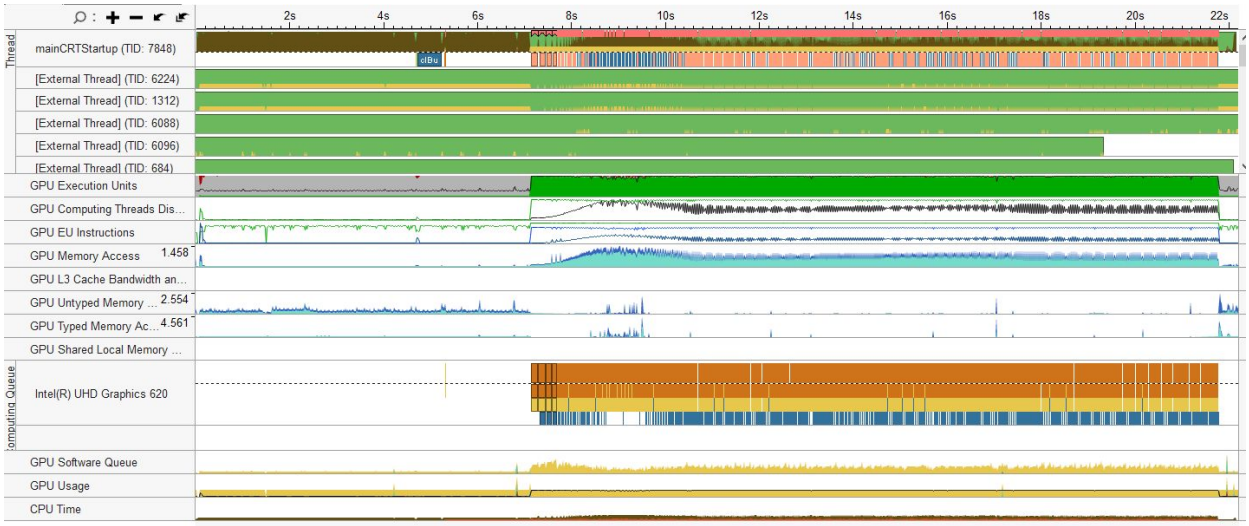
GPU Hotspot Analysis Summary

Computing Task Purpose / Source Computing Task (GPU)	Computing Task			Data Transferred		EU Array			EU Threads Occupancy	EU Instructions		
	Total Time ▼	Average Time	Instance Count	Size	Total, GB/sec	Active	Stalled	Idle		IPC Rate	2 FPUs active	Send active
▸ Compute	14.091s	0.035s	400		0.000	97.8%	2.0%	0.1%	99.5%	1.813	79.4%	0.4%

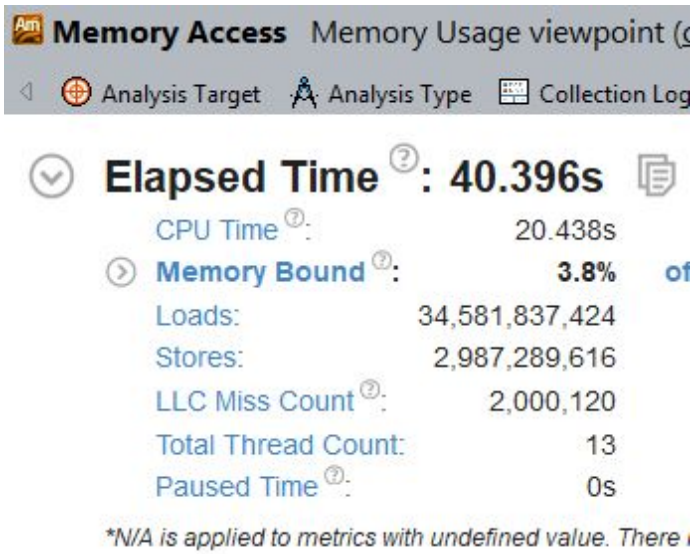
GPU Hotspot Analysis Compute Metrics

Computing Task Purpose / Source Computing Task (GPU)	L3 Shader Bandwidth, GB/sec	Memory Transactions Coalescence				Shared Local Me...	
		Untyped Reads	Untyped Writes	Typed Reads	Typed Writes	Read	Write
▸ Compute	4.891	0.0%	0.0%	0.0%	0.0%	0.000	0.000

GPU Hotspot Analysis Memory Metrics



GPU Hotspot Analysis Graphics



Summary of Memory Access

With all of the pictures above, you can see clearly that my kernel is compute bound. The kernel is constantly filling up the queue and running. In the top picture, the 65% GPU usage is misleading, as it takes 8 seconds before the program runs the kernel. Once the kernel starts running, it uses up as many resources as it can handle. This is most easily seen through the EU Array active statistic, while the kernel is running, it is active 97.8% of the time. The Thread occupancy tells a similar story with a 99.5% occupancy rate. While inside any individual kernel, the floating point utilization sits at 79.4%. There is still room for improvement, but any improvement would need to come from a more efficient algorithm, rather than keeping the GPU busy.

As for memory, the kernel only accesses global memory. Because each kernel is operating on a very small set of read data, and only writes 3 chars, the use of local memory

does not seem to make sense. Each kernel reads in 3 floats, and 6 chars, while writing out 3 chars. With such a small memory footprint, there is no memory bandwidth problem.

Measured Speedup

For my measured speed up test, I decided to do use a 512x512 image, max iterations of 100, going from order 1.0 to 11.0, at a step size of .1. Under these conditions the Serial implementation took 122.311 seconds to run. The OpenCL implementation took 1.667 seconds. This is a speed up of 73x, which is not nearly as much as the theoretical speedup.

Discussion

For this project, I started out with a lofty goal of producing a fractal generating kernel that could run smoothly in real-time. Over the course of my implementation I first got a working product then began with the slow process of making improvements. There were 4 major sections of development that increased the quality of my output. Serial implementation, OpenCL implementation, kernel optimizations, and host optimizations.

I began implementation where I was most comfortable, in C++ doing a serial implementation. By copying some implementations from the internet, I was able to get a barebones implementation going at a pretty quick rate. During this time, I learned three things, what the implementation was actually doing, how to write the image to data, and how to display the image to the screen while running in real-time. Each of these sections was important in my end goal of running this on the GPU. I needed to know what the kernel was doing if I was to implement it, I needed to know how to get the data to and from the kernel, and I needed to take that data and display it in a valid format.

The next step was to take my serial implementation and transform it into an OpenCL kernel. The first thing I did, was produce a C++ version of the host API. This allowed me to work where I was comfortable, and keep the files I was working with shorter. Once I had a stable and working host program, I started work on the kernel. The first step was to copy the inner loop of my serial implementation and put it into a kernel and find what sections needed to be rewritten. This copying showed me two things, relying too heavily on C++ methods makes transfering to C only hard. Specifically in regards to using the complex libraries that C++ provides. In C, there is no complex standard library that could be used by the kernel. Thankfully, the developers of OpenCL provide a library that does complex math. Once I had this library in my hands, my kernel ran. I was able to update the images in real-time, but they had frame rates of only 5 FPS.

Trying to speed up my kernel, I did my first round of performance analysis. In this analysis, I found that nearly half of the time that my kernel ran for was spent transferring data to and from the kernel. My first pass implementation required the kernel to save its entire state every run and pass that information to the next kernel that ran. This transfer of data took a long time, but it gave the kernel some flexibility in its functionality. Realizing this, I decided to write my kernel out on the fly instead of having one more complex kernel. By removing the need to have one kernel that could do it all, I opened up the opportunity of tuning the kernel I wanted to run to do only exactly what I wanted. By cutting down the transfer of data to the kernel, I increased the throughput of my kernel. I was able to achieve a frame rate of ~10 FSP.

The final improvements to my program came in the form of host changes. With the kernel running at near maximum capacity, while it was active, I needed to look to the host to find new ways to improve the throughput of the program. I implemented three tricks that drastically improved the throughput of the kernel. The first trick was to stop waiting for the kernel to finish,

but to instead wait for when its data was ready to use on the host side. This small change increased the throughput by nearly 3-5 FPS. This is because anytime the command queue is empty, it takes a bit of time for it to get back up to speed. So, enqueueing the buffer map command right after the kernel command, the data was ready for processing right when clFinish exited. The second improvement came by optimizing the kernel's local worksize. I optimized the work size by running through all valid local sizes with a small kernel and choosing the one that ran the fastest. This bumped my frame rate by another 3-5 FPS. The third improvement came when I saw that the CPU was waiting uselessly while the kernel ran. I needed a way to have the kernel always running, as its computation took much longer than any CPU process. The solution was to double buffer my kernel enqueues. By double buffering my kernel enqueues I had two streams of data that were always being worked on. While one kernel was processing data, the CPU was working on the data from the second stream. This improvement improved the frame rate by ~15-20 FPS.

Overall, the kernel that I implemented efficiently uses GPU & CPU resources simultaneously to produce interesting visuals. There are still improvements to make, and new modes to implement, but the first application of this kernel is a success.

References

https://en.wikipedia.org/wiki/Fractal
https://www.geeksforgeeks.org/fractals-in-cc/
https://www.libsdl.org/
https://github.com/OpenCL/ComplexMath